CS422 - Programming Language Design

Elements of Functional Programming

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

The two languages that we defined so far in the class, namely the simple imperative language that we defined in several definitional frameworks and λ_{K} that we used for introducing K, were toy languages; their role was to present the various language definitional styles, not to show prototypical examples of languages.

In this class we will define several more complex languages, language features, and paradigms.

Our next goal is to design and define modularly a *functional language*, that we will call `FUN`. In this lecture we discuss the basic features that we want to include in `FUN`. These features are standard in many functional languages, such as *OCAML*, *ML*, *Scheme*, *Haskell*, and so on.

Our purpose is *not* to define any of these known functional languages, though you will be able to do it easily at the end of the course, but rather to define their major features in a modular way, so that one can create a new language by just combining these feature modules. FUN, as well as any other language that we will define in this class using K, will therefore be easily configurable.

It is very important to first understand the concepts underlying the language that you want to design. Without the big picture in mind, your design can be poor and non-modular, so difficult to change. Today's lecture is dedicated to understanding the language we want to define, FUN.

Functional Programming Languages

Functional programming languages are characterized by allowing *functions as first class citizens*. This means that functions are manipulated like any other values in the language, so in particular they can be passed as arguments to functions, can be returned by functions, and so on.

The syntax of functional languages is typically very simple, but there are various, usually non-trivial, semantic choices when one designs a functional programming language. Syntactically, almost everything is an *expression*. Expressions are *evaluated* to *values*. As we did with the simple languages that we previously defined, we start with expressions that can be built with integers, identifiers and common operators, which we assume already defined.

Let

The `let <Bindings> in <Exp>` construct is encountered in most, if not all, functional programming languages. Its meaning is essentially to bind some names to values and then to evaluate an expression which may refer to those names. For example,

```
let x = 5
in x
```

is a new expression, which is evaluated to 5. One can have multiple bindings:

```
let x = 5
and y = 7
in x + y
```

Nested `let` expressions are naturally allowed:

```
let x = 5
in let y = x
   in y

let x = 1
in let z = let y = x + 4
           in y
   in z
```

Both expressions above should evaluate to 5. The meaning of the `let` language construct in a given state is the following:

> Evaluate the expression in the `in` part in the state obtained after evaluating all the right-hand-side expressions in the bindings and then assigning their values to the corresponding names *in parallel*.

Notice that nothing is specified about the *order* in which the right-hand-side expressions are evaluated! Because of *side effects*, which we will allow in our language, different orders can lead to different behaviors. Different implementations (or models) of our language can take different decisions; one can even evaluate all the expressions concurrently on a multiprocessor platform.

Also, it is important to note that the right-hand-side expressions are evaluated *before* the bindings are applied. The following expression, for example, is evaluated to whatever value `x` has in the current state, which may be different from 10:

```
let x = 10 and y = 0 and z = x
in let a = 5 and b = 7
   in z
```

To keep our language simpler to parse, we are going to use a slightly modified syntax for `let` (and also for `letrec` that will be discussed shortly) in our K executable definition of `FUN`, namely `let(Xl, El, E)`, where `Xl` is a list of variables (the binding variables), `El` is a list of expressions (the expressions bound to variables, and `E` is an expression (the body): the variables `Xl` will be bound to the expressions `El`, respectively, and then `E` will be evaluated in the obtained environment. With this syntax, the expression above is written admittedly more ugly as:

```
let (( x, y, z), (10, 0, x),
  let ((a, b), (5, 7),
    z
  )
)
```

Exercise 1 *Change the K definition of* `FUN` *defined in the next lecture to accept the more readable syntax* `let X1 = E1 and ... and Xn = En in E`.

Hint: *Define a list sort* `Bindings` *with operations*
`_=_ : Var Exp -> Bindings`,
`_and_ : Bindings Bindings -> Bindings`, *and*
`(_,_) : VarList ExpList -> Bindings`,
and with equations/rules (in fact, only one suffices!) collapsing terms `X1 = E1 and ... and Xn = En` *into terms* `((X1,...,Xn),(E1,...,En))`; *once variables and expressions are gathered each kind together, then one can use almost the same definitions for* `let` *and* `letrec` *as we already have in the current definition of* `FUN`.

Functions

Functions stay at the core of any functional language, so they will also play a crucial role in FUN. Similarly to many other functional languages, we will use a syntax of the form `fun <Parameters> -> <Exp>` to write functions, and syntactically they are nothing but ordinary expressions.

Different functional languages may use different `function` keyword and general syntax for function definitions, such as "`fn <VarList> => <Exp>`", or "`lambda (Var, Exp)`", or "`lambda Var Exp`", etc.; all these minor syntactic differences are ultimately irrelevant. In FUN, the following is a well-formed expression defining a function:

```
fun (x, y, z) -> x * (y - z)
```

If a function has only one argument, then we take a freedom to not enclose that argument between parentheses. For example, the following parses correctly:

```
fun x -> x * (x - 1)
```

A function is allowed to have *no parameters*. The empty sequence of parameters is written simply `()`, and it is called the *unit*. Thus, the following is a correct function (`x`, `y`, and `z` are expected to be declared in the outer environment):

```
fun () -> x * (y - z)
```

Function Application

To apply functions, we need to pass them a corresponding number of arguments. To pass arguments to functions, we need an operation `<Exp> <ExpList>`, called *function application*, or *invocation*, whose first argument is expected to evaluate to a function. We will assume the application operation to be left associative, to eliminate the need of parentheses. The following are therefore all well-formed expressions:

```
(fun (x,y) -> 0) (2,3)
(fun (y,z) -> y + 5 * z) (1,2)
(fun y -> fun z -> y + 5 * z) 1 2
```

The first applies a function with two arguments which returns `0`, the second applies a more complicated function, and the third is a "curried" version of the second (currying is discussed shortly) and

shows a function whose result is another function, so it can be applied again. The expected values after evaluating these expressions are, of course, `0`, `11` and again `11`, respectively. Note how the third expression evaluates; due to the left-associativity of function application, it parses to

```
((fun y -> fun z -> y + 5 * z) 1) 2
```

and is expected to first evaluate to

```
(fun z -> 1 + 5 * z) 2
```

and then eventually to evaluate to `11`.

Optional Material: Currying

An important observation, which is at the core of many functional languages, is that a function with several arguments can be seen as a series of nested functions. More precisely,

```
fun (x, y, z) -> x * (y - z)
```

is entirely equivalent to

```
fun x -> fun y -> fun z -> x * (y - z)
```

This technique of transforming a function taking multiple arguments into a function that takes a single argument, namely the first of the arguments to the original function, and returns a new function which takes the remainder of the arguments and returns the result, is called *currying*. The technique is named after logician Haskell Curry, though it was invented by other scientists before.

Thanks to currying, multiple-argument function declarations may

be regarded as just *syntactic sugar* in many functional languages. For that reason, one only needs to define function application for single argument functions. In order to do this, we only need an operation `<Exp> <Exp>`, whose first argument is expected to evaluate to a function (as opposed to an operation `<Exp> <ExpList>` as we use in our definition of `FUN`).

Exercise 2 *Change the K definition of* `FUN` *that will be discussed next to define the semantics of functions via currying. In other words, define the function application to take only a one-expression (second) argument and then curry the multiple arguments.*
Hint. *For currying, only one equation is needed.*

Static Type Checking

Static *type checkers*, discussed and defined later in the course, ensure that functions are applied correctly. For example, a type checker will forbid expressions of the form

```
(fun (y,z) -> y + 5 * z) 1
((fun y -> fun z -> y + 5 * z) 1) + 2
```

For the time being, we allow such wrongly typed expressions to be correctly parsed as expressions. However, we'll be able to catch such errors both dynamically and statically later in the course.

Binding Functions

One may want to bind a name to a function in order to reuse it without typing it again. This can be easily done with the existing language constructs:

```
let f = fun (y,z) -> y + 5 * z
in  f(1,2) + f(3,4)
```

Evaluating the above expression should yield 34. To simplify writing and readability of FUN programs, like in other major functional languages (e.g., OCAML), we allow another syntactic sugar convention: bindings of functions of the form <name> = fun <Parameters> -> <Exp> can be written more compactly as <name> <Parameters> = <Exp>. For example, the expression above can be written:

```
let f(x,y) = y + 5 * z
in  f(1,2) + f(3,4)
```

Passing Functions as Arguments

In functional programming languages in general, and in FUN in particular, functions can be passed as arguments to functions just like any other expressions. E.g.,

```
(fun (x,y) -> x y) (fun z -> 2 * z) 3
```

should evaluate to 6, since the outermost function applies its first argument, which is a function, to its second argument.

Similarly, the following evaluates to 1:

```
let f(x,y) = x + y
and g(x,y) = x * y
and h(x,y,a,b) = x(a,b) - y(a,b)
in h(f,g,1,2)
```

Free vs. Bound Names

Like we had uninitialized variables in programs written in the simple language described in the previous lecture, we can also have free names in expressions.

Intuitively, a name is *free* in an expression if and only if that name is *referred to* in some subexpression without being apriori *declared* or *bound* by a `let` construct or by a function parameter. E.g., `x` is free in the following expressions:

```
x
let y = 10 in x
let y = x in 10
x + (let x = 10 in x)
let x = 10 and y = x in x + y
```

as well as in the expressions

```
fun y -> x
fun () -> x
fun y -> y + x
(fun y -> y + 1) x
(fun y -> y + 1) 2 + x
x 1
(fun x -> x) x
```

A name can be therefore free in an expression even though it has several bound occurrences. However, `x` is *not free* in any of the following expressions:

```
let x = 1 in (x + (let x = 10 in x))
fun x -> x
let x = fun x -> x in x x
```

Scope of a Name

The same name can be declared and referred to multiple times in an expression. E.g., the following are both correct and evaluate to 5:

```
let x = 4
in let x = x + 1
   in x

let x = 1
in let x = let x = x + 4 in x
   in x
```

A name declaration can be thus *shadowed* by other declarations of the same name. Then for an occurrence of a name in an expression, how can we say to which declaration it refers to? Informally, the *scope* of a declaration is "the part of the expression" in which any occurrence of the declared name refers to *that* declaration.

Static vs. Dynamic Scoping (or Binding)

Scoping of a name is trickier than it seems, because the informal "part of the expression" above cannot always be easily defined. What are the values obtained after evaluating the following?

```
let y = 1
in let f(x) = y
   in let y = 2
      in f(0)
```

```
let y = 1
in (fun (x,y) -> x y) (fun x -> y) 2
```

To answer this question, we should first answer the related question "what declarations of `y` do the expressions `fun x -> y` refer to?". There is no definite answer, however, to this question.

Under *static (or lexical) scoping*, it refers to `y = 1`, because this is the most nested declaration of `y` containing the occurrence of `y` in `fun x -> y`. Thus, the scope of `y` in `fun x -> y` can be determined statically, by just analyzing the text of the expression. Therefore, under static scoping, the expressions above evaluate to 1.

Under *dynamic scoping*, the declaration of `y` to which its occurrence in `fun x -> y` refers cannot be detected statically anymore. It is a dynamic property, which is determined during the evaluation of the expression. More precisely, it refers to the latest declaration of `y` that takes place during the evaluation of the expression. Under dynamic scoping, both expressions above evaluate to 2.

Most of the programming languages in current use prefer static scoping of variables or names. Software developers and analysis tools can understand and reason about programs more easily under static scoping. However, dynamic scoping tends to be easier to implement. There are languages, like GNU's BC, which are

dynamically scoped. The very first versions of LISP were also dynamically scoped.

Since both types of scoping make sense, in order to attain a maximum of flexibility in the design of our programming language, we will define them as separate Maude modules and import whichever one we want when we put together all the features in a fully functional language.

It is important to be aware of this design decision all the time during the process of defining our language, because it will influence several other design decisions that we will make.

Functions Under Static Scoping

Under static scoping, all the names which occur free in a function declaration refer to statically known previous declarations. It may be quite possible that the names which occurred free in that function's declaration are redeclared before the function is invoked.

Therefore, when a function is invoked under static scoping, it is *wrong* to just evaluate the body of the function in the current state (that's what one should do under dynamic scoping)! What one should do is to *freeze* the state in which the function was declared, and then to evaluate the body of the function in *that state* rather than in the current state.

In the context of side effects the situation will actually be more complicated, since one actually wants to propagate the side effects across invocations of functions. In order to properly accommodate

side effects, the *environments* when the functions are declared rather than the states will be frozen; environments map names to *locations*, which further contain values, instead of directly to values.

This special value keeping both the function and its declaration state or environment is called a *closure* in the literature. We will discuss this concept in depth in subsequent lectures, and define it rigorously when we define our `FUN` language.

But for the time being, think of a closure as containing all the information needed in order to invoke a function. It is a closure that one gets after evaluating a function expression, so closures are seen as special values in our language design.

Static Scoping and Recursion

Is there anything wrong with the following expression calculating the factorial of a number recursively?

```
let f(n) = if n eq 0
           then 1
           else n * f(n - 1)
in f(5)
```

There is nothing wrong with it under dynamic scoping, because once `f(5)` starts being evaluated, the value denoted by `f` is already known and so will stay when its body will be evaluated.

However, under static scoping, the `f` in `f(n - 1)` is not part of the closure associated to `f` by the `let` construct, so `f(n - 1)` *cannot* be evaluated when the function is invoked.

Letrec

Therefore, in order to define recursive functions under static scoping we need a new language construct. This is called "`letrec`", possibly written also using two words, namely "`let rec`", and is supported by many functional programming languages. For example, that would be the definition of factorial:

```
letrec f(n) = if n eq 0
              then 1
              else n * f(n - 1)
in f(5)
```

It can be used to also define mutually recursive functions:

```
letrec 'even(x) = if x eq 0 then 1 else  'odd(x - 1)
and     'odd(x) = if x eq 0 then 0 else 'even(x - 1)
in 'odd(17)
```

Unlike `let`, which first evaluates the expressions in its bindings, then creates the bindings of names to the corresponding values, and then evaluates its body expression in the new state, `letrec` works as follows:

1. Creates bindings of its names to currently unspecified values, which will become concrete values later at step 3;
2. Evaluates the binding expressions in the newly obtained state;
3. Replaces the undefined values at step 1 by the corresponding values obtained at step 2, thus obtaining a new state;
4. Evaluates its body in the new state obtained at step 3.

If one does not use the names bound by `letrec` in any of the binding expressions then it is easy to see that it is behaviorally equivalent to `let`.

However, it is crucial to note that those names bound using `letrec`

are accessible in the expressions they are bound to! Those of these names which are bound to function expressions will be therefore bound to closures including their binding in the state.

More precisely, if `S'` is the new state obtained at step 3 above when `letrec` is evaluated in a state `S`, then the value associated to a name `X` in `S'` is

- The value of `X` in `S` if `X` is not a name bound by `letrec`;
- A closure whose state (or environment, in the context of side effects) is `S'` (or the environment of `S'`, respectively) if `X` is bound to a function expression by `letrec` or to an expression which evaluates to a function;
- An integer or an undefined value otherwise.

While this is exactly what we want in the context of recursive functions, one should be very careful when one declares non-functional bindings with `letrec`. For example, the behavior of

the expression

```
let x = 1
in letrec x = 7
   and    y = x
   in y
```

is undefined. Notice that the variable `x` is *not free* in

```
letrec x = 7
and    y = x
in y
```

Instead, it is bound to a location which contains a value which is not yet defined!

Variable Assignment

So far our functional programming language was *pure*, in the sense that it had *no side effects*. More precisely, this means that the value associated to any name in a state did not change after evaluating any expression.

Indeed, if a name is redeclared by a `let` or `letrec` construct, then a new binding is created for that name, the previous one remaining untouched. For example, the expression below evaluates to `1`:

```
let x = 1
in let y = let x = x + 4 in x
   in x
```

The evaluation of the expression `let x = x + 4 in x` to `5` has no effect therefore on the value of `x` in the outer `let`.

There are situations, however, when one wants to modify an

existing binding. For example, suppose that one wants to define a function `f` which returns the number of times it has been called. Thus, `f() + f()` would be evaluated to `3`. In typical programming languages, this would be realized by defining some global variable which is incremented in the body of the function. In our current language, the best one can do would be something like

```
let c = 0
in let f() =  let c = c + 1
              in c
   in f() + f()
```

or

```
let f = let c = 0
        in fun () -> let c = c + 1
                     in c
in f() + f()
```

Unfortunately, neither of these solves the problem correctly, they

both evaluate to `2`. The reason is that the `let` construct in the body of the function *creates a new binding* of `c` each time the function is called, so the outer `c` will never be modified.

By contrast, a *variable assignment* modifies an existing binding, more precisely the one in whose scope the assignment statement takes place. Following most functional programming languages, we allow variable assignments in `FUN`. We let `<Name> := <Exp>` denote a variable assignment expression. Like any expression, a variable assignment expression also evaluates to a value, which by convention will be *unit*. This value is of almost no importance: variable assignments are essentially used for their side effects.

With this, the two expressions above can be correctly modified to the following, where `d` is just a dummy name used to enforce the evaluation of the variable assignment expression:

```
let c = 0
in let f() = let d = c := c + 1
             in c
   in f() + f()
```

and

```
let f = let c = 0
        in fun () -> let d = c := c + 1
                     in c
in f() + f()
```

which evaluate to 3.

In order to properly handle and define side effects, and in particular variable assignments, in a programming language, one has to refine the state by splitting it into *environment* and *store*.

The environment maps names to *locations*, while the store maps locations to values. Thus, in order to extract the value associated

to a name in a state, one first has to find that name's location in the environment and then extract the value stored at that location.

All language constructs can be defined smoothly and elegantly now. `let` creates new locations for the bound names; assignments modify the values already existing in the store; closures freeze the environments in which functions are declared rather than the entire states. Side effects can be now correctly handled.

Parameter Passing Variations

Once one decides to allow side effects in a programming language, one also needs to decide how argument expressions are passed to functions. So far, whenever a function was invoked, our intuition was that bindings of its parameters were created to new values obtained *after evaluating* the expressions passed as arguments. This kind of argument passing is known as *call-by-value*.

Under call-by-value, the following expression evaluates to 2:

```
let x = 0
in let f x = let d = x := x + 1
             in x
   in f(x) + f(x)
```

Other kinds of parameter passing can be encountered in other programming languages and can be quite useful in practice.

Suppose for example that one wants to declare a function which swaps the values bound to two names. One natural way to do it would be like in the following expression:

```
let x = 0 and y = 1
in let f x y = let t = x
               in let d = x := y
                  in let d = y := t
                     in 0
   in let d = f x y
      in x + 2 * y
```

However, this does not work under call-by-value parameter passing: the above evaluates to 2 instead of 1. In order for the above to work, one should *not* create bindings for function's parameters to new values obtained after evaluating its arguments, but instead to bind functions' parameters to the already existing locations to which its arguments are bounded. This way, both the argument

names and the parameter names of the function after invocation are bound to the same location, so whatever new value is assigned to one of these is assigned to the other too. This kind of parameter passing is known as *call-by-reference*.

One natural question to ask here is what to do if a function's parameters are call-by-reference and when the function is invoked it is passed (as arguments) expressions *that are not names*. A language design decision needs to be taken. One possibility would be to generate a runtime error. Another possibility, which is the one that we will consider in our design, would be to automatically convert the calling type of those arguments to call-by-value.

Another important kind of parameter passing is *call-by-need*. Under call-by-need, an argument expression is evaluated only if needed. A typical example of call-by-need is the conditional. Suppose that one wants to define a conditional function `cond` with three arguments expected to evaluate to integers, which returns either its third or

its second argument, depending on whether its first argument evaluates to zero or not. The following defines such a function:

```
let x = 0 and y = 3 and z = 4 and
    'cond(a,b,c) = if a eq 0 then c else b
in 'cond(x, (y / x), z)
```

Like in the expression above, there are situations when one does *not* want to evaluate the arguments of a function at invocation time. In this example, `y / x` would produce a runtime error if `x` is `0`. However, the intended role of the conditional is exactly to avoid evaluating `y / x` if `x` is `0`. There is no way to avoid a runtime error under call-by-value or call-by-reference.

Under call-by-need, the arguments of `cond` are bound to its parameter names *unevaluated* and then evaluated only when their values are needed during the evaluation of `cond`'s body. In the situation above, `a`, `b` and `c` are bound to `x`, `y / x` and `z`, all unevaluated (or *frozen*), respectively, and then the body of `cond` is

evaluated. When `a eq 0` is encountered, the expression bound to `a`, that is `x`, is evaluated to `0`; this value now *replaces* the previous binding of `a` for later potential use. Then, by the semantics of `it_then_else` which will be soon defined formally, `c` needs to be evaluated. It's value, `4`, replaces the previous binding of `c` and it is then returned as the result of `cond`'s invocation. The expression `y / z` is never needed, so it stays unevaluated, thus avoiding the undesired runtime error.

Call-by-need parameter passing is also known as *lazy evaluation*. One can arguably claim that call-by-need is computationally better in practice than call-by-value, because each argument of a function is evaluated *at most once*, while under call-by-value all arguments are evaluated regardless of whether they are needed or not. There are important functional programming languages, like *Haskell*, whose parameter passing style is call-by-need. However, since one does not know how and when the arguments of functions are

evaluated, call-by-need parameter passing is typically problematic in program analysis or verification.

The fourth parameter passing style that we will consider is *call-by-name*, which differs from call-by-need in that the argument expressions are evaluated each time they are used. In the lack of side effects, call-by-need and call-by-name are behaviorally equivalent, though call-by-need is more efficient because it avoids re-evaluating the same expressions. However, if side effects are present, then call-by-name generates corresponding side effects whenever an argument is encountered, while call-by-need generates the side effects only once.

We will define only call-by-value in class. However, as a homework exercise, you will define the other three styles of parameter passing as well. You will define them in separate modules, and include only those which are desired in each particular programming language design. In order to distinguish them, each parameter will be

preceded by its passing-style, e.g., `fun(val x, ref y, need z, val u, name t) -> ...`.

Sequential Composition and Loops

One way to obtain sequential composition of statements is by using `let` constructs and dummy names. For example, `Exp1` followed by `Exp2` followed by `Exp3` can be realized by

```
let d = Exp1
in let d = Exp2
   in Exp3
```

The dummy name must not occur free in any of the sequentialized expressions except the first one. Sequential composition makes sense only in the context of side effects. For example, the expression

```
let d = x := x + y
in let d = y := x - y
   in let d = x := x - y
      in Exp
```

occurring in a context where `x` and `y` are already declared, evaluates to `Exp` evaluated in a state in which the values bound to `x` and `y` are swapped.

Since side effects are crucial to almost all useful programming languages and since sequential composition is a basic construct in these languages, we will also define it in `FUN`.

More precisely, we will define a language construct {`<ExpList>`}, where `<ExpList>` is a list of expressions separated by semicolons, whose meaning is that the last expression in the list is evaluated in the state obtained after propagating all the side effects obtained by evaluating the previous ones sequentially.

The above then can be written using the more common syntax:

```
{ x := x + y ;
  y := x - y ;
  x := x - y ;
  Exp }
```

Exercise 3 *What are the values to which the following two expressions evaluate?*

```
let f(need x) = x + x
in let y = 5
   in {
        f(y := y + 3) ;
        y
      }

let y = 5
and f(need x) = x + x
and g(ref x) = x := x + 3
in {
     f(g(y));
     y
   }
```

What if we change the parameter passing style to call-by-name?

Like sequential composition, *loops* do not add any computational power to our already existing programming language either, because they can be methodologically replaced by recursive functions defined using `letrec`. However, since loops are so frequently used in almost any programming language and are considered basic in most algorithms, `FUN` will also provide them.

Like in the simple imperative language defined in the previous lecture, `FUN` will have both `while` and `for` loops. Their syntax will be `while <Exp> <Exp>` and `for(<Exp>;<Exp>;<Exp>)<Exp>`. They will be just other expressions, having the expected meaning.

We will also allow to `break` and `continue` loops. In class we will discuss the nonparametric variants; as a homework exercise you will define the *parametric* `break` and `continue`.

Then, in FUN, the Collatz conjecture program looks as follows:

```
let n = 178378342647 and c = 0
in {
     while not (n eq 1) {
       c := c + 1 ;
       if n eq 2 * (n / 2)
       then n := n / 2
       else n := 3 * n + 1
     } ;
     c
   }
```